Lecture notes: Replication for fault tolerance in the Cloud

M2 MOSIG: Cloud Computing, from infrastructure to applications

Thomas Ropars

2025

This lecture is about techniques based on replication to avoid service failures in the Cloud. More specifically, it presents techniques that can be applied to deal with benign faults[1].

1 Introduction

1.1 Fault model

To be able to reason about fault tolerance techniques, we need to define an abstract model that captures the main characteristics of the faults we want be able to tackle.

In the following, we will assume **benign** faults. For processes, benign faults correspond to crashes: a process either executes according to the specification, or stops executing. For channels, benign faults correspond to the loss of messages.

Another model that could have been considered is **Random faults (Byzantine)**. In this model, a system provides *random* outputs to the same inputs. It covers the cases where different users might observe different behaviors and the cases of malicious behaviors. Due to the complexity of the solutions that deal with byzantine faults, we do not consider them hereafter.

1.2 Introduction to replication

The basic way of interacting between services/processes in Clouds and data centers is through a client/server model. A client sends a request to another node and waits for an answer.

Replication is a technique that allows us to increase the availability of a system or service. With replication, instead of having only one instance of a system/service, there are several copies. So, if one copy crashes, the system/service will still be available, thanks to the other copies (Fig. 1).

Several approaches to replication exists. We can distinguish 3 main categories:

Data replication : With this approach, the client can only issue read and write operations on the data. On the other hand, it allows a high degree of parallelism.

Passive replication : This approach is also called *active/standby* replication. A single leader executes all requests and sends updates to the other replicas.

[1]Acknowledgments: Parts of these notes are strongly inspired by the lectures notes of Andre Schiper on *Distributed Algorithms*.

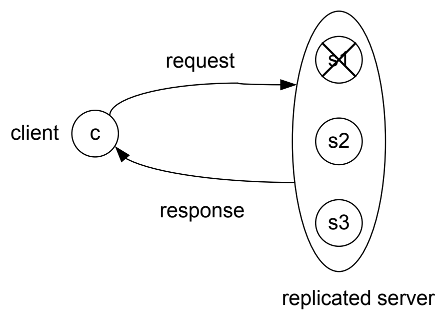

Figure 1: Replication (client-server model)

Active replication : This approach is also called *active/active* replication, or multi-leader replication. In this approach, all replicas can process requests.

Note that availability might not be the only concern when using replication. Replication may also be used to improve performance:

- To reduce the latency by keeping data geographically close to the users
- To improve throughput by allowing parallel reads

These concerns should also be taken into account when choosing a replication technique.

2 Data replication (Quorum systems)

In a first step, we consider a simple problem where data need to be replicated. Only two operations can be applied on a data item, *read* and *write*:

- `write` overwrites the previous value of the data item,
- `read` returns the most recent value written.

Defining the most recent value written might not be that trivial when the data is replicated on multiple servers. Figure 2 illustrates a basic scenario. We need a consistency criteria to define what is an acceptable result for a read operation.

2.1 Linearizability

The strongest consistency criteria for defining what is an acceptable result for a read request is called *linearizability.*

Definition Let x be a data item, and denote the *read* operation of x by $read(x)$, and the *write* operation by $write(x, val)$, where val is the value written. Consider an execution σ consisting of the concurrent execution of read and write operations by a set of processes. The execution σ is *linearizable* if there exists a *sequential* execution τ in which:

(1) All read operations in τ return the same value as in σ

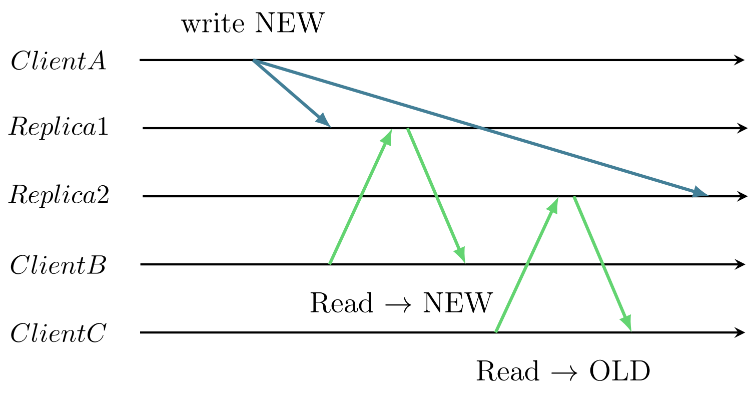

Figure 2: A problematic scenario

and

(2a) The start and the end of each read or write operation op in τ occur at the *same* time, denoted by t^{op}, and

(2b) The time t^{op} is in the interval $[t_s^{op}, t_e^{op}]$, where t_s^{op} is the time at which the operation op started in σ and t_e^{op} is the time at which the operation op ended in σ.

Example 1: Figure 3 shows an execution σ that is linearizable:

- Process p executes a $write(x, 0)$ operation that starts at time t_1 and ends at time t_3.
- Process q executes a $write(x, 1)$ operation that starts at time t_2 and ends at time t_5.
- Process p executes a $read(x)$ operation that starts at time t_4 and ends at time t_7, returning the value 0.
- Process q executes a $read(x)$ operation that starts at time t_6 and ends at time t_8, returning the value 1.

The bottom time-line in Figure 3 shows a sequential execution τ that satisfies the three conditions *(1)*, *(2a)* and *(2b)* above (t_a is in $[t_1, t_3]$, t_b is in $[t_4, t_7]$, etc.). Thus σ is linearizable.

Note that to define *linearizability*, we consider the operations for the client point of view. Figure 3 does not describe how clients p and q interacts with the (replicated) server.

Example 2: Figure 4 shows an execution σ that is not linearizable. The execution is not linearizable, since in any sequential execution τ, the $write(x, 1)$ operation of q must precede the $read(x)$ operation of p. Thus in any sequential execution τ, the $read(x)$ operation of p returns 1 (and not 0 as in σ).

Informal definition of linearizability Informally, we can say that linearizability aims at making the system appear as if there was a single copy of the data. To achieve this goal, we should enforce the following behavior:

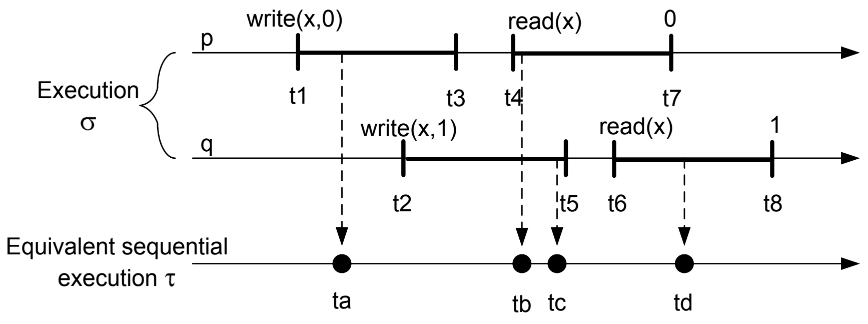

Figure 3: A linearizable execution

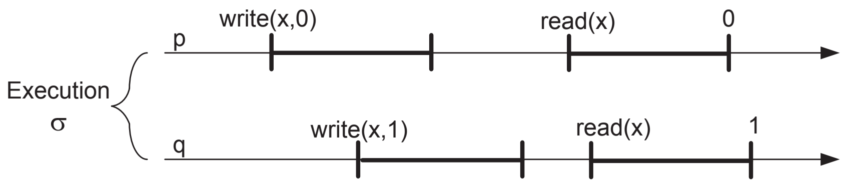

Figure 4: A non linearizable execution

- If a read request terminates before the beginning of a write request, it must return the old value.
- If a read request starts after the end of a write request, it must return the new value.
- If a read request is concurrent with a write request, it may return either the old or the new value.

In addition to these three basic rules, we should introduce the following one:

- After a read request that returned the new value, all subsequent read requests, from the same or from a different client, must return the new value.

This last rule describes the fact that in a linearizable system, a write operation should appear to occur atomically at some point between the invocation of the write method and its termination.

The definition of linearizability assumes the existence of a global clock to be able to define if a request occurred before/after another one, based on the physical time. Building such a clock may be difficult in practice but it is not necessary to have one to build a linearizable system.

Linearizability in practice Linearizability is a very desirable property because it simplifies the work of the application developers. Indeed, linearizability is a composable property. It means that if an application uses two linearizable services, the resulting execution is still linearizable. It is not the case for weaker consistency models (such as sequential consistency).

Unfortunately, insuring linearizability can be costly from performance point of view. Hence, many services make the choice to implement a weaker consistency model. We will resume this discussion later.

2.2 Non-replicated implementation of a linearizable data server

A non replicated server is easy to implement. The server consists of one process that manages the data. Clients send requests and wait to receive a reply from the server (this is called *synchronous invocation*). Requests are received and handled by the server process sequentially. A $write(x, val)$ request by client p leads the server to update x to val, and to send ok to p. A $read(x)$ request by client p leads the server to send val to p. This implementation trivially ensures linearizability.

2.3 Replicated implementation of a data server

In this section, we are going to present a replication technique that is sometimes called *leaderless replication*. In such an approach, a client directly contacts one/several replicas to execute read or write operations. This name is given by opposition to *leader-based* approaches (that we will study later), where the client sends its requests to a single node that is in charge of synchronizing with the other replicas.

2.3.1 Quorum systems

To build a linearizable replicated data server, we are going to rely on the concept of *quorum system*. Consider a set S of n servers $S_1, \dots S_n$, i.e., $S = \{S_1, \dots, S_n\}$:

- A *quorum system* of S is a set of "subsets of S", such that any two "subsets of S" have a non empty intersection.

For example, if $S = \{S_1, S_2, S_3\}$, then

$$Q = \{ \ \{S_1\}, \{S_1, S_2, S_3\} \ \}$$

is a quorum system of S. The following set Q' is also a quorum system of S:

$$Q' = \{ \ \{S_1, S_2\}, \{S_1, S_3\}, \{S_2, S_3\} \ \}$$

Each element of Q, and each element of Q', is called a *quorum*. Note that each quorum of Q' contains a majority of servers.

With quorum systems, the basic idea can be expressed as follows. Consider a quorum system of S:

- Each write operation must update a quorum of servers.
- Each read operation must access a quorum of servers.

As such, it is guaranteed that each read request will retrieve the latest value that has been written.

Another way of presenting the idea of quorum systems is to say that if there are n replicas, each write must update w replicas and each read must access r replicas, with $w + r > n$. In a system with 3 replicas ($n = 3$), it is very common to set $w = r = 2$, which corresponds to the quorum system Q'. However a system may be configured differently depending on the needs. For instance, a system could be configured with $w = 3$ and $r = 1$ to have very fast reads at the cost of slower writes. Another limitation of this configuration is that if one server fails, then all write operations will fail.

2.3.2 Fault tolerance with a quorum system

To ensure fault tolerance, we should rely on a majority rule for quorums for both read and write operations. A majority rule states that $r > n/2$ and $w > n/2$. With such a rule, one is able to tolerate up to f server crashes, with $f < n/2$.

Here are a few additional comments about quorums and fault tolerance:

- If $w < n$, we can still write if a node crashes
- If $r < n$, we can still read if a node crashes
- A configuration with $n = 3$, $w = r = 2$, can tolerate on crash
- A configuration with $n = 5$, $w = r = 3$, can tolerate two crashes

It should also be mentioned that even if w and r are configured to be less than n, read and write requests are usually sent to all replicas by the client. The value of w and r defines the number of answers to wait for before considering an operation as terminated.

Based on the majority rule, if we consider the 2 quorum systems Q and Q' defined earlier, we can conclude that Q' is an appropriate quorum system for fault tolerance. Q is not.

2.3.3 Linearizable data replication based on quorum systems

We present a solution to implement linearizability for data items in a system with at most f faults ($f < n/2$).

Server code: The code executed by one replica S_i is presented in Figure 5. As it can be noticed, in addition to the value of the data item, two additional information are maintained by the server: a version number and the identifier of the client that most recently updated the item.

```
value = 0 # data item value
version = 0 # version number
clientId = 0 # id of the most recent client that has written the item

upon readReplica() by client $c$:
    send (value,version,clientId) to $c$

upon writeReplica(val, v, id): #$v$ is a version number; $id$ is the client id
    if (v > version) or ((v==version) and (id > clientId)):
        value = val
        version = v
        clientId = id
```

Figure 5: Read and write operation: code of a replica S_i

The version number should be incremented every time a client updates the data item. The information about the `clientId` is needed to deal with concurrent writes. As shown in line 9, the `writeReplica` operation overwrites the current value only if: i) the version number of the operation is higher than the current version number or, ii) the version numbers are equal and the identifier of the client is larger than the one of the last client that updated the item.

Client code (first try): Figure 6 presents a first version of the client algorithm. The `write` operation first requires to read from a quorum (line 14). Only the `version` number is a relevant information in this case: the read operation is needed to obtain a valid version number. The client issues the write to a quorum with a version number incremented by one (line 16).

The `read` operation issues a read from a quorum and returns the value associated with the highest {`version`, `clientId`} observed.

```
13      def write(val, id):                    # id is the identifier of the client
14          readReplica(--, version, --) from a quorum
15          v = highest version number read
16          writeReplica(val, v+1, id) to a quorum        # synchronous invocation

18      def read():
19          readReplica(val, version , id) from a quorum
20          let (v, id) be the highest (version number, client Id) read
21          let value be the value with version/clientId (v, id)
22          return value
```

Figure 6: Code of a client C (Incorrect version)

The simple algorithm presented in Figure 6 is unfortunately not correct.

It does not ensure a linearizable execution as demonstrated by the scenario presented in Figure 7. In this scenario, linearizability would require the read operation of `Reader2` to return the new value (or `Reader1` to return the old value).

Client code (final version): To solve this problem, we should implement a technique called *read repair*. During a read operation, if a client observes stale values, it should synchronously issue a write to a quorum with the most recent value observed before terminating the read operation.

The new version of the client code is provided in Figure 8. The only modification compared to the algorithm presented in Figure 6 is the call to `writeReplica()` during the read operation (line 32).

The result is that, instead of the execution depicted in Figure 9 that corresponds to the scenario presented in Figure 7 and which is obviously a non-linearizable execution, we can get the execution depicted in Figure 10, which is linearizable.

2.3.4 Replication based on quorum systems in practice

Replication based on quorum systems is used by several NoSQL databases. Dynamo, the scalable KV-store used internally at Amazon, is one of the systems that made this approach popular. Apache

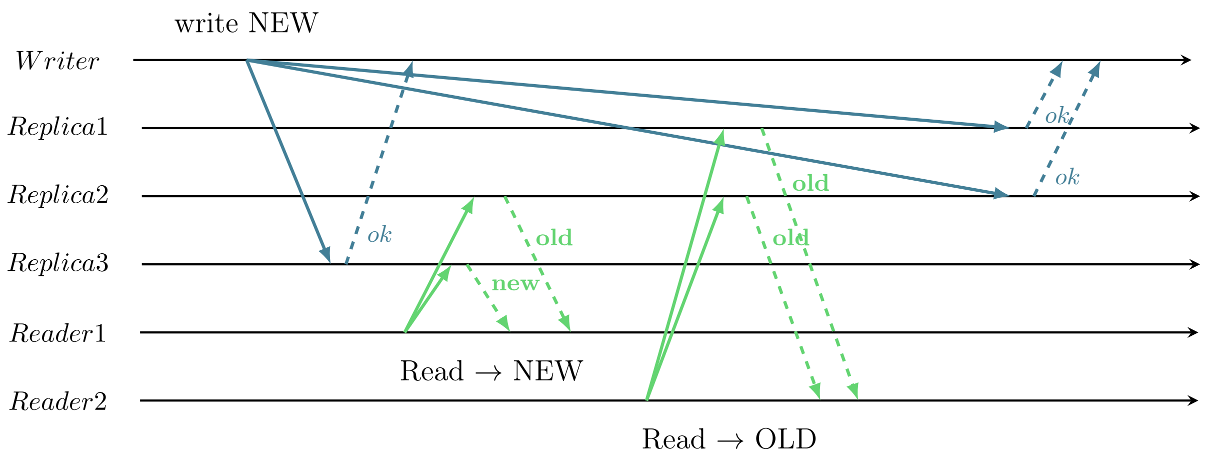

Figure 7: A non-linearizable execution with quorums

```
def write(val, id):                  # $id$ is the identifier of the client
    readReplica(--, $version$, --) from a quorum
    v = highest $version$ number read
    writeReplica(val, v+1, id) to a quorum       # synchronous invocation

def read():
    readReplica(val, version , id) from a quorum
    let $(v, id)$ be the highest (version number, client Id) read
    let $value$ be the value with version/clientId $(v, id)$
    writeReplica(value, v, id) to a quorum       # synchronous invocation
    return $value$
```

Figure 8: Code of a client C (Correct version)

Cassandra[2] is a famous open-source database that also uses this approach. When applying quorum-based approaches in practice, additional problems arise. We discuss some of them below.

Read repair and anti-entropy The *read repair* technique introduced in Algorithm 8 is not only used to ensure linearizability. It is also used to update a replica that has a stale version of the data because it experienced a failure. However, this mechanism slows down read operations. Furthermore, if a data is not accessed very often, it may take time before a stale replica is updated. As long as the replica is not updated, fault tolerance is reduced as the number of valid copies of the replica is decreased.

To deal with both issues, some databases introduce an *anti-entropy* mechanism. Anti-entropy is a background task that checks the health of data, and applies updates to stale replicas if need be.

Dealing with concurrent writes If two clients decide to write the same data item *at the same time*, a quorum-based algorithm needs to decide what should be the final value to store. Imagine that 2 clients, `A` and `B`, send a write request concurrently. Because of network transmission delays

[2]`http://cassandra.apache.org/`

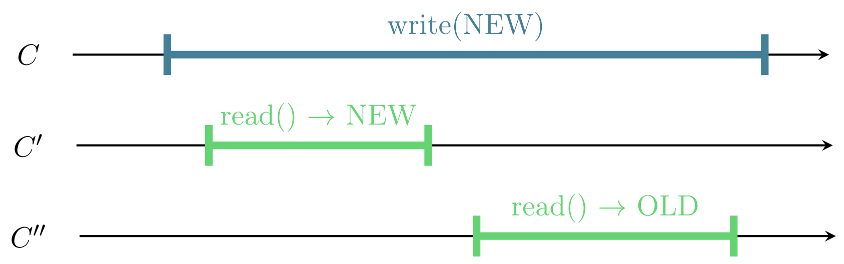

Figure 9: A possible execution with algorithm 6

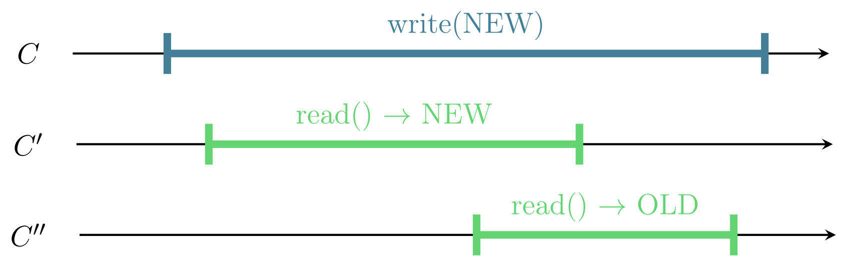

Figure 10: A possible execution with algorithm 8

(among other things), it may be the case that not all replicas receive the requests in the same order. In a system with 3 replicas, 2 may receive the request of `A` first while the last one will receive the one of `B`. In this situation, we need a solution to ensure that all replicas will select the same value as final value.

In Algorithm 5, the solution relies on the identifier of the clients to take a decision: the write operation issued by the client having the largest identifier wins. It solves the problem but we can ask ourselves whether it makes sense that client `B` has the priority over client `A`.

An alternative approach, adopted for instance by Cassandra, is to assign a timestamp to each request, and to decide than the most recent request will win. This approach is called *last-write-wins* (LWW). Such a solution can be better for fairness. As long as the clock of the different clients remain *well* synchronized, such a solution can work. However, it might lead to non-linearizable executions if clock skews are large. Discussing such a scenario in more details is beyond the scope of this lecture.

About multi-datacenter configurations To increase the availability of data and services and/or to reduce the latency of data accesses, some cloud applications are distributed over geographically distant data centers. Although very desirable, linearizability might be difficult to achieve in such a context for two main reasons.

The first reason is the limited availability that can be achieved. Imagine a scenario where replicas of a data item are distributed over two data centers DC_a and DC_b. Due to a network failure, communication between the two data centers is temporarily not possible: clients trying to access the data are only able to contact the replicas in one data center (can be DC_a or DC_b depending on the client). The majority rule that we apply to ensure linearizability despite failures implies that only a subset of the clients will be able to continue accessing the data. Indeed, either the majority of replicas is in DC_a or in DC_b. If we assume that it is in DC_a, it means that the clients only having access to DC_b cannot read or write the data anymore, and so, the availability is reduced.

To deal with this issue, a technique called *sloppy quorums* may be used. The idea is to create more replicas inside a data center when w replicas are not accessible anymore from a client. This avoids that write operations will fail. Once the connection between the datacenters is restored, the additional replicas are destroyed, and the modifications in the different partitions are merged. Even if such a solution can ensure the durability of write operations, it leads to non-linearizable executions as read and write quorums may not overlap anymore.

The second reason is performance. As implied by the previous discussion, ensuring linearizability in a multi-datacenter setup requires that at least some clients access a remote replica, *i.e.*, a replica that is not in the closest datacenter, to follow the majority-quorum rules. In this case, it means that read and write operations may become slow as they involve potentially high latency communication. Once again, the solution in this case is to lower the level of consistency to avoid having to communicate with remote datacenters synchronously.

3 Single-leader replication

The quorum-based approach discussed previously considers data items with two operations: read and write. We are now studying a more general problem where an *object* is replicated, and where more complex update operations on the object can be applied.

3.1 Linearizability for objects

The definition of linearizability for data items can be repeated for general objects.

An execution σ consisting of the concurrent execution of a set of processes is *linearizable* if there exists a sequential execution τ in which

(1) All operations in τ that return a result return the same value as in σ,

and

(2a) The start and the end of each operation *op* in τ occurs at the *same* time, denoted by t^{op}, and
(2b) The time t^{op} is in the interval $[t_s^{op}, t_e^{op}]$, where t_s^{op} is the time at which the operation *op* started in σ and t_e^{op} is the time at which the operation *op* ended in σ.

3.2 Quorum systems for general objects?

To illustrate why quorum-based approaches do not work for general objects, we consider a replicated *counter* object with two operations, *increment* and *decrement*. The two operations update the counter.

Assume that *increment* and *decrement* are implemented by (1) reading the counter followed by (2) writing the new value into the counter. If two clients c and c' execute *increment* concurrently, the following can happen:

- c and c' get the same version number (line 25 of Algorithm 8) and the same value *val* of the counter;

- c and c' write $val+1$ as the new value (line 26 of Algorithm 8). While two increment operations are requested, only one of them takes effect!

Mutual exclusion is needed here, as illustrated in Figure 11. However, mutual exclusion is tricky to solve with process crashes. If a client c crashes while in the critical section, the mutual exclusion privilege must be taken from c. This require a failure detection algorithm to decide when to remove privilege from c. The issue is that mutual exclusion may be compromised if c is wrongly suspected of being failed.

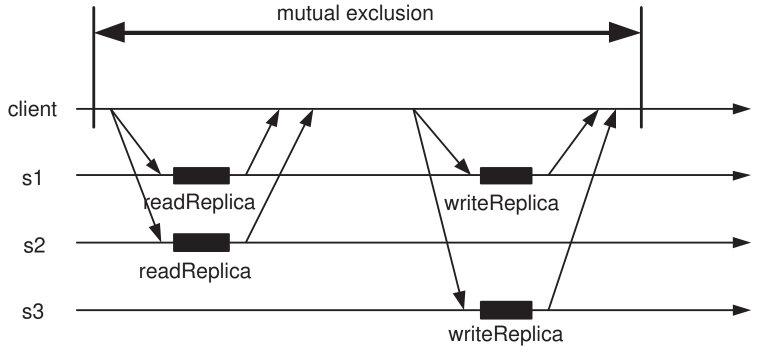

Figure 11: Mutual exclusion for replicated objects

One may wonder why, in the above example, the operations *increment* and *decrement* are not sent to the servers (instead of decomposing these operations using read and write operations). This is precisely what will be done, but this is more difficult than it appears at a first look.

3.3 Single-leader replication

Single-leader replication is also sometimes called *passive* replication (or *active/passive* replication, or *primary/backup* replication). In this approach, a single leader replica receives the requests from the client, applies the corresponding operations, and sends updates to the other replicas (that are hence passive). Figure 12 illustrates the basic communication scheme.

Such a replication strategy is adopted by many systems including relational databases (*e.g.*, MySQL), non-relational databases (*e.g.*, MongoDB), and message brokers (*e.g.*, Kafka).

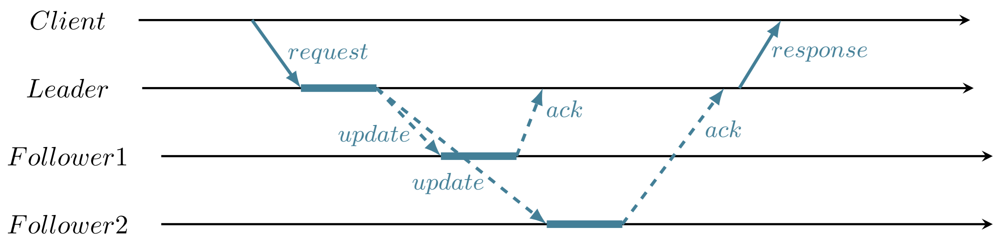

Figure 12: Single-leader replication protocol

In this approach, all operations that require a modification of the data have to be executed through the leader. However, if one of the goal of replication is to improve the performance of the system, the solution is to allow clients to access any replica for read-only operations. Such a solution allows serving read request more efficiently, but it has an impact on the consistency. We will come back to this point soon.

3.3.1 Synchronous and asynchronous updates

In Figure 12, updates are synchronously propagated by the leader to the other replicas. The leader waits for an acknowledgment from all the followers that the update has been applied before sending a response to the client.

With synchronous replication, if the leader crashes, the other replicas have an up-to-date copy of the object. However, synchronous replication has drawbacks:

- It can be significantly impacted by delays. If a replica becomes slow or if some communication are delayed, the duration of an operation increases from the point of view of the client.
- It reduces the availability of the system. If a replica crashes, we have to wait until it restarts to be able to complete some update operations.

An alternative is to use asynchronous replication: the leader does not wait for an acknowledgment that the update has been applied on the other replicas before answering to the client. Hence, asynchronous replication can be much more efficient than synchronous replication. However, it has the drawback that updates might be lost even after an acknowledgment has been sent to the client. It means that the system becomes less fault tolerant and that it provides only limited consistency guarantees.

In practice, when a system provides synchronous replication, it usually means that one replica is updated synchronously while the other replicas are updated asynchronously, as illustrated in Figure 13. This guarantees that we have an up-to-date copy of the data on at least two nodes, without paying the price for fully synchronous updates.

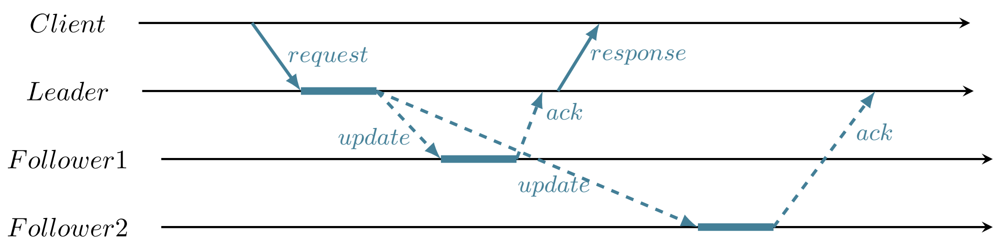

Figure 13: Single-leader configuration with one synchronous replica

3.3.2 Implementation of the replication mechanisms

In the implementation of leader-based replication, one major point to be discussed is about the information included in the update messages sent by the leader to the other replicas. Two of the main approaches are the following:

Sequence of requests: The leader directly sends the requests to be executed to the followers that should apply them in the same order as the leader. One limitation of such an approach is that if the execution of a request involves non-deterministic actions, then the replicas might not be consistent anymore (which implies that such an approach cannot be used in this case).

State updates: The leader executes the requests and sends updates that are modifications of the object state to the followers. The updates may be captured in different ways. It may be a log of all the modifications applied to the state at byte granularity. It may be a higher level description of the writes applied to the state (`e.g.`, modification of a row in a relational database). Here, non-determinism is not an issue anymore, as the followers do not re-execute the requests.

The two approaches can be compared from efficiency point of view. There is no clear winner. If the execution of requests is very CPU intensive, a state-update approach might be more efficient.

3.3.3 Managing failures

We focus on the failure on the leader, as it has a specific role in this replication technique. The process of reconfiguring the system to work with a new leader is called *failover*. Here are the main steps involved:

Failure detection: The first step is to detect that the leader has crashed. Accurate failure detection is challenging in cloud environments. Using long timeouts decreases the risk of false positives at the expense of a lower availability of the system.

Selecting a new leader: Having all replicas agreeing on a new leader boils down to solving a consensus problem (which should be handled carefully in a distributed system). The new leader may be selected taking into account which follower has the more up-to-date version of the data.

System reconfiguration: The clients needs to learn about the new leader to send the requests to the right destination.

Many things can go wrong during the failover process. Because failure detection might not be accurate or because the old leader restarted after a crash, we might end up having two leaders in the system. This case needs to be handled. Also, some updates might be lost in a failover process, because the elected leader does not start from the most recent object state (the risk of loosing updates is higher with asynchronous replication).

3.3.4 About consistency

For a single-leader replication protocol to provide linearizability, all requests (including read requests) have to be processed sequentially by the leader. However, if replication is used to improve the performance of read operations, any replica should be able to serve read requests. In this case linearizability cannot be ensured. We illustrate the problem in Figure 14 assuming asynchronous replication (but the problem also exists with synchronous replication).

Figure 14 illustrates the fact that clients might observe inconsistencies. However, if we stop issuing new update operations, and if the leader does not crash, the replicas will eventually become consistent again. We say that such a system provides *eventual consistency*.

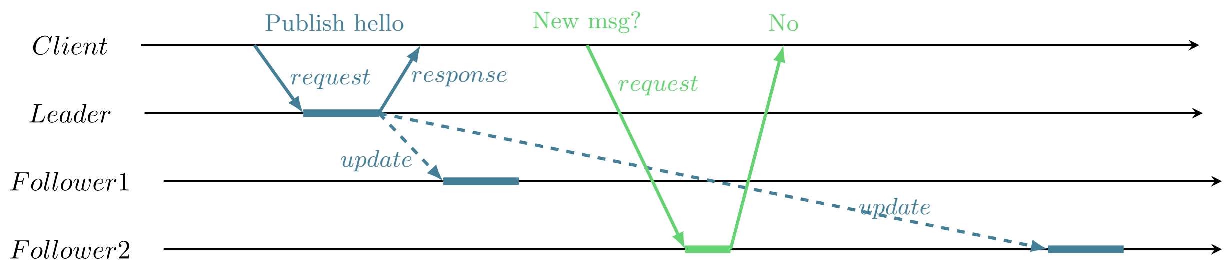

Figure 14: Single-leader non-linearizable execution

Eventual consistency can work well in practice because most often followers will be updated in a very short time. Hence, the probability of a client observing inconsistencies is small.

In some cases, we might still want to provide consistency guarantees that are a bit stronger. For instance, we might want to ensure that the scenario presented in Figure 14 does not occur, that is, guaranteeing that a client sees its own writes. This is called *read-after-write* consistency. Here are two possible solutions to ensure *read-after-write* consistency:

- We can force some requests to go through the leader. These requests are the one that could access objects that the user may modify. In a social network, a user may modify her profile. Hence, all requests implying reads of the user own profile should go through the leader to ensure read-after-write consistency.
- If a user can edit a majority of the objects, the previous approach is not efficient. An alternative is to rely on time (preferably logical time). The idea is to remember the time of the most recent update issued by a client, for instance using a sequence number, and to ensure that the state of the replica serving a read request includes at least the corresponding updates.

4 Multi-leader replication

Multi-leader replication (also called *active* replication or *active/active* replication) is a replication approach where the clients can contact any replica to execute their read and update requests.

Obviously, such an approach raises additional challenges with respect to consistency. In a first step, we will discuss solutions that ensure linearizability. Then we will discuss about solutions providing weaker guarantees.

4.1 Solutions that provide linearizability

Multi-leader replications strategies that provide linearizability need to ensure that despite the fact that any replica may receive requests to execute, all replicas should execute the same requests in the same order. To ensure this, they rely on a communication primitive that is called *total order broadcast* (or *atomic broadcast*).

We can informally define[3] the properties of total order broadcast as follow:

- All correct processes deliver the same set of messages.

[3]Formally defining total order broadcast is outside the scope of this course

- Any message is delivered at most once by a process.
- Any message broadcast by a correct process is eventually delivered by all correct processes.
- All processes deliver the messages in the same order.

To ensure linearizability with multi-leader replication, it is enough to ensure that each request to be executed is first total-order broadcast in the set of replicas, and to execute the requests in the order they are total-order delivered (this is called state-machine replication). Such an approach assumes that the execution of requests is deterministic.

Note that solving total order broadcast in a distributed environment is equivalent to solving consensus. It is thus a difficult problem.

4.2 Multi-leader replication and weak consistency

In most cases, in a setup where all replicas are executed in the same data center, multi-leader replication does not provide any advantage over single-leader replication. Since it is simpler to implement, single-leader replication is most often used in this case.

However, in a setup where replicas are distributed over multiple data centers, multi-leader replication can become interesting.

Obviously, and as discussed earlier, linearizability is too costly to ensure in a multi-cloud setup. Instead, we have to go with weaker consistency guarantees.

The two main advantages of multi-cloud environments are: i) a performance increase by bringing the data closer to the users; ii) a higher availability by being able to survive to the crash of one data center. However, to fully take advantage of multi-cloud deployments, any replica should be able to process requests locally. This is what multi-leader replication should allow.

Allowing multiple replicas to apply updates locally raises a new problem: conflicting writes may be executed. Indeed, if two replicas execute updates that modify the same data, we later need to resolve this write conflict.

One may want to resolve conflicts synchronously, *i.e.* synchronizing with the other replicas to resolve conflicts before terminating an update operation. In this case, we will mostly lose the advantages of using multi-leader replication for a multi-cloud setup. Hence, write conflicts have to be solved asynchronously.

Different approaches might be used to deal with conflicting writes:

- *Conflict avoidance:* If one can ensure that any two modifications of the same part of the application state goes through the same leader, then write conflicts will be avoided
- *Defining priorities for updates:* techniques such as last-write wins or client with higher identifier wins may be applied.
- *Custom conflict resolution logic:* Some systems allow the programmer to define a callback that is going to be executed to resolve conflicts when one is detected. Such a callback may be called during a write operation that generates a conflict or during a read operation that accesses a data for which a conflict has been detected.

It should also be mentioned that collaborative editing is a case of multi-leader replication. If users are allowed to edit shared documents offline, this is an extreme case for asynchronous write conflicts resolution.

To go further

Some references can complement the material presented in these lecture notes:

- Chapter 5 of *Designing Data-Intensive Applications* by Martin Kleppmann
- Section *Linearizability* in Chapter 9 of *Designing Data-Intensive Applications* by Martin Kleppmann

Fault tolerance through replication for Cloud systems is an active research topic, with many important new publications every year. To complement this course, among the papers published recently, we can mention [1] that illustrates the fact that building fault tolerant systems that can always handle fault correctly is difficult: Considering 25 widely used fault-tolerant software, the authors study 104 scenarios that lead to catastrophic failures. Regarding eventual consistency versus stronger consistency guarantees, a study of a very larger scale eventually consistent system shows that in almost all cases, it provides the same results as a linearizable system [2].

References

[1] A. Alquraan et al. An analysis of network-partitioning failures in cloud systems. In *OSDI*, 2018.

[2] H. Lu et al. Existential consistency: measuring and understanding consistency at facebook. In *SOSP*, 2015.